Nicholas Chan
nipchan@ucsc.edu
5/23/2021
CSE13s Spring ‘21

Design Document:
Assignment 6: Huffman Coding

Huffman coding is a system for compressing data based on the concept of entropy. The entropy or measure of chaos within a file can be represented through the construction of a histogram covering all the permutations of a uint8_t (there will be 256 entries in the histogram/array). Based on the distribution of frequencies among uint8_t characters, less frequent uint8_t’s receive a code with more bits while more frequent uint8_t’s receive a code with less bits. With a constructed histogram we can create a Huffman Tree where each unique character is assigned as a leaf node. The positions of each leaf node in the tree give their corresponding characters a compressed code that can be written out. The decoding process for Huffman coding involves the reconstruction and traversal of the Huffman tree to extract the characters corresponding to encoded inputs.

In assignment 6, I have been given the task of implementing an encoder, decoder and IO module for a huffman coding interface as well as ADTs for nodes, codes, priority queues and stacks. Encoding is a process which will require the construction of a Huffman tree from nodes. A priority queue will be used to order the nodes before they are dequeued and added to the Huffman tree. Characters will be encoded based on their positions in the tree.

Top Level:
Pseudocode/functionality

Overview

- Encoding:
  - Form Histogram
    - Array of 256 uint64_t items
    - Indices are characters, values frequencies
  - Construct tree from Node ADT and histogram info
    - Tree building requires Priority Queue (Uses Node ADT)
    - Tree building functions in Huffman Coding Module (Uses Priority Queue to build the Huffman Tree)
  - Form code table from Code ADT
  - Write out encoded/compressed characters with I/O module functions
- Decoding (Reconstructing and traversing the Huffman Tree):
  - Check for correct magic number from input file
  - Reconstruct Huffman tree from input file tree dump
    - Reconstructed using Stacks (Uses Node ADT)
    - See rebuild tree rom Huffman Coding Module (No Priority queue)
  - Read input to traverse reconstructed tree for writing decoded characters

Node ADT

- The node struct contains
  - Node pointer to left
  - Node pointer to right
  - uint8_t symbol
  - uint64_t frequency
- node_join takes in pointers to left and right nodes, returns a node pointe
  - Create new node named n
  - Symbol of n = $
  - Frequency of n = sum(left and right freqs)
  - Node to the left of n = left
  - Node to the right of n = right
- Usual ADT functions
  - Create
    - Requires input for symbol and frequency
  - Delete
  - Print

Priority Queue PQ ADT (Similar to regular queue)

- PQ struct contains
  - uint32_t for Capacity
  - uint32_t for Size

    - uint32_t for Head
    - uint32_t for Tail
- Enqueue takes in a priority queue pointer q and a node pointer n to enqueue and returns a bool
    - Create a placeholder index called tmp
    - While tmp != head
        - If frequency of node pq[tmp-1] >frequency of node n
            - Copy node pq[tmp-1] over to node pq[tmp]
            - tmp -= 1
        - Else
            - Node pq[tmp] is set to node n
- Dequeue takes in a priority queue pointer q and a pointer to a node pointer n to enqueue and returns a bool
    - Decrements queue size and assigns node to the input node pointer
    - Return true on a successful dequeue, false o.w.
- Usual ADT functions
    - Create
        - Requires input for capacity
    - Delete
    - Full
    - Empty
    - Print

Code Module (All functions except init take a pointer)

- Code struct contains:
    - uint32_t for top
        - Used to get the top bit like in the stack struct
    - uint8_t array of byte sized items
        - Size of array = MAX_CODE_SIZE macro
        - The bitvector (behaves almost exactly like bitvector)
- code_init is the constructor of Code instances
    - Create Code called c
    - Set top of c = 0
    - Return pointer to Code c
- code_size returns the size of a code given a Code pointer
- code _push_bit
    - Push a bit (1 or 0) onto the top of the code bit vector
    - Return true if bit was successfully pushed, false o.w.
- code _push_bit
    - Pop a bit from the top of the code bit vector
    - Return true if bit was successfully popped, false o.w.
- code_print
    - Print code contents

Huffman Coding Module

- build_tree
  - Take histogram array as input and calculate total of unique ascii characters
  - Create priority queue of size = total unique ascii characters
  - Enqueue a node to the priority queue for each unique ascii character and its corresponding frequency
  - Repeat the process of double dequeue, joining node, enqueue node to priority queue until 1 node is left in priority queue
  - Return final node (now the root of the Huffman tree)
- build_codes
  - Perform a postorder traversal on a root node (Huffman tree) and assign codes to characters based on their placements on each leaf node.
  - A left traversal adds a bit of 0 and a right traversal adds a bit of 1 to the code
- delete_tree
  - Deletes tree with a postorder traversal on the root node
- Rebuild_tree
  - Reconstructs tree given tree dump sequence and the length of the tree dump sequence
  - Iterate through tree dump and push nodes generated from tree dump components to a stack
  - Pop stack nodes in a similar fashion as in build_codes to reconstruct the Huffman tree

I/O System Calls/Module

- read(file descriptor, buffer, number of bytes) and write(file descriptor, buffer, number of bytes)
  - man read() and write()
  - Takes in buffer, number of bytes and file descriptor
  - Buffer size is specified as the BLOCK macro (4096 bytes)
- read_bytes
  - Perform a read on infile to a buffer of size block for a specified number of bytes
  - Return the total number of bytes read in
- write_bytes
  - Perform a write to outfile from a specified buffer for a specified number of bytes
  - Return the total number of bytes written out
- read_bit
  - Fills a buffer and passes back bits read in from bytes in buffer
  - Returns true for a successful bit pass and false o.w.
- write_code
  - Writes a specified code to outfile
  - Performs bitwise operations on items in an array while keeping track of bit indices
- flush_codes

- Writes out leftover bytes in buffer + remainder byte if necessary

Stack

- Contains
  - Integer indicating top position
  - Integer indicating capacity
  - Pointer to an array of node pointers
- Stack_push (takes in stack pointer and node pointer)
  - Push node to a stack
  - Increment top of stack by 1
  - Return true for a successful push, false otherwise
- Stack_pop (takes in stack pointer and a pointer to a node pointer)
  - Pop node from stack and assign node pointer to the pointer to a node pointer from input
  - Decrement top of stack by 1
  - Return true for a successful pop, false otherwise
- Usual ADT functions
  - Delete stack
  - Return Empty status
  - Return Full status
  - Return Size
  - Print

Design Process

I felt that the first thing I needed to understand for this assignment was its use of nodes, tree construction and codes. I began by constructing the node ADT and manually joining them together until I moved onto joining them through means of enqueuing and dequeuing from priority queues. Then I constructed codes from the positions of leaf nodes with postorder traversals and bit operations. I then moved onto writing out the Header contents, tree dump and encoded messages with the operations in the IO module. I thought the IO module was the most challenging part of this assignment as output was not as easy to understand. The output from my encoder also turned out to be different than the resource encoder's output, but the addition of my header contents and tree dump allowed them to function the same. Moving onto the decoding portion of the assignment, reading in the header contents and tree dump were a slightly less challenging process. I was able to reconstruct the Huffman tree from the tree dump and proceeded to decode the buffered input by bits. Decoded messages were added to a BLOCK sized buffer and written out when the buffer filled. Below is a more ordered sequence of events on how I approached this assignment.

1. The first ADT I constructed was the Node

    a. This node struct consisted of a symbol (using uint8_t) and a uint64_t value for symbol frequency
    b. Nodes may be connected to a node on either its left or right sides via pointer assignments in the struct
    c. Nodes can be joined
        i. Useful for constructing trees
2. The main algorithms utilized for encoding messages and decoding bit codes were tree constructions and postorder traversals.
    a. Tree constructions used information on the number of unique ascii characters from input and their frequencies
        i. A tree would be the top most node from a network of connected nodes
        ii. Trees could be constructed using a sequence enqueues, dequeues and node_joins on the node items of a priority queue
            1. The priority queue would prioritize nodes with lower frequencies to a placement near the front
    b. A code table could be constructed from performing postorder traversals on a constructed Huffman tree (root node) until all symbols of leaf nodes were assigned a corresponding code
        i. A left traversal would add a 0 to the code of a symbol while a right traversal would add a 1 to the code of a symbol
            1. Each travel down adds a code bit while each traversal up subtracts a code bit
    c. The tree dump and Header contents written on output give the decoder the information necessary for reconstructing the Huffman tree and ultimately the ability to decode compressed codes
        i. Contents are written out via write_byte functions and write_code functions from the IO module.
3. The decoder utilitzes read_bytes from the IO module to receive Header file information like the valid magic number for decoding a file, the input file's original size and the size of the original huffman tree
    a. The read_bits function is also called to fill a BLOCK sized buffer with compressed codes where it can decipher them bit by bit using a tree traversal of a reconstructed huffman tree
        i. read_bits makes a call on read_bytes for reading in data
        ii. Huffman tree in decode is reconstructed given the tree dump from input and the tree size from the Header file contents
        iii. Decoded messages are stored on a BLOCK sized buffer in the decoder file where they are written out according to the original uncompressed size of the input file (down to the byte)

4. All statistics are kept track of by the bytes_read and bytes_written variables which are kept in the IO module files
    a. They are referenced in the encoder and decoder for the verbose function as well as keeping track of the number of bytes to write and read.

I thought this assignment was pretty challenging especially with the need for so many IO calls. The use of read,write and buffers were a little hard for me to grasp, but Eugene's and Sahiti's sections really helped clear them up for me. I was also able to experience how buffering output speeds up the writing process. For the decoder I originally had a write call every time a message was decoded, but that was very inefficient (my runtime for the bib file under Calgary in corpora was ~30 seconds). I tried buffering the decoded messages to a BLOCK sized buffer and writing became much much faster. This experience really pounded in the fact that system calls are expensive.

References:

- https://www.youtube.com/watch?v=joX93VhNlRo